Swendsen-Wang Multi-Cluster Algorithm for the 2D/3D Ising Model on Xeon Phi and GPU

Florian Wende
Zuse Institute Berlin
Takustrasse 7
D-14195 Berlin-Dahlem
wende@zib.de

Thomas Steinke
Zuse Institute Berlin
Takustrasse 7
D-14195 Berlin-Dahlem
steinke@zib.de

ABSTRACT

Simulations of the critical Ising model by means of local update algorithms suffer from critical slowing down. One way to partially compensate for the influence of this phenomenon on the runtime of simulations is using increasingly faster and parallel computer hardware. Another approach is using algorithms that do not suffer from critical slowing down, such as cluster algorithms. This paper reports on the Swendsen-Wang multi-cluster algorithm on Intel Xeon Phi coprocessor 5110P, Nvidia Tesla M2090 GPU, and x86 multi-core CPU. We present shared memory versions of the said algorithm for the simulation of the two- and three-dimensional Ising model. We use a combination of local cluster search and global label reduction by means of atomic hardware primitives. Further, we describe an MPI version of the algorithm on Xeon Phi and CPU, respectively. Significant performance improvements over known implementations of the Swendsen-Wang algorithm are demonstrated.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]:
Concurrent Programming—*Parallel Programming*;
I.6.8 [**SIMULATION AND MODELING**]:
Types of Simulation—*Monte Carlo, Parallel*;
J.2 [**PHYSICAL SCIENCES AND ENGINEERING**]: Physics

General Terms

ALGORITHMS, PERFORMANCE

Keywords

Many-core processors, Xeon Phi, GPGPU, CUDA, Ising model, Swendsen-Wang multi-cluster algorithm, performance evaluation, graph algorithms

1. INTRODUCTION

The present paper focuses on the evaluation of the compute performance of current parallel processor platforms—Intel Xeon Phi hardware accelerator, Nvidia Tesla M2090 GPU, and Intel octa-core Xeon E5-2680 CPU, namely—for the simulation of the two- and the three-dimensional critical Ising model, which serves as a test model for algorithm design and performance evaluation, and as a toy model in statistical physics for already several decades. In particular we report about the parallelization of Swendsen and Wang's multi-cluster algorithm for the Ising model, which, amongst the single-cluster algorithm by U. Wolff, is the matter of choice for simulations at criticality.

At the core of the Swendsen-Wang algorithm is connected component labeling which has application domains in Percolation Theory, Computer Vision to detect connected regions in images, and Computational Physics, to name a few.

The structure of the paper is as follows: Section 2 recaps work already done in simulating the Ising model on parallel computer systems, and outlines this paper's contributions in this field. Section 3 gives a brief overview of the hardware used in this paper. In Sec. 4, we introduce the Ising model. Our approach to simulating the Ising model by means of the Swendsen-Wang multi-cluster algorithm on parallel processor platforms is detailed in Sec. 5, including an MPI version. Section 6 summarizes benchmark results and measurement data. Concluding remarks are given in Sec. 7.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13, November 17-21, 2013, Denver, CO, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503254

2. RELATED WORK

Since its introduction and the advent of commodity computer systems, the Ising model has been studied intensively by means of simulation programs using vector processors, FPGAs, and multi-core processors, respectively. The use of GPUs for that purpose became increasingly popular during the past years. Since GPUs are usually used for the execution of programs with massive data parallelism, the major focus of investigations in the context of GPU-based simulations of spin models was on local update algorithms of Metropolis type [24, 25, 26]. With the newer GPU generations, porting (non-local) cluster algorithms to these devices became feasible. Cluster update algorithms for the simulation of the two-dimensional Ising spin model on GPU are detailed in [12, 23]. Y. Komura and Y. Okabe achieve update times per spin down to 2.86ns on a single Nvidia GeForce GTX580 GPU [13] respectively 0.029ns on a multi-GPU system made up of 256 Nvidia Tesla M2050 devices [14].

The present paper extends these investigations by the following aspects:

- We give implementations of the Swendsen-Wang cluster algorithm for the two- and the three-dimensional Ising model. To our knowledge, this paper is the first that reports on parallel implementations of the said algorithm for three dimensions on GPU, and for two and three dimensions on Xeon Phi.

- We describe a novel label reduction technique using atomic hardware primitives.

- We present an MPI version of the Swendsen-Wang algorithm suitable for simulations on lattices with more than $2^{32}-1$ sites despite 32-bit words are used for the labeling.

3. AN OVERVIEW OF THE PROCESSOR PLATFORMS

Table 1 summarizes the processor platforms considered in this paper. Access to a Xeon Phi cluster was generously provided to us by Intel.

3.1 Intel Many Integrated Core (MIC) Architecture

The Xeon Phi consists of more than 50 Pentium-P54C-based cores, each of which augmented with 64-bit support, fully-coherent L1 and L2 cache, 32 512-bit SIMD registers, and 4-way interleaved hardware multi-threading [8, 9, 11]. With its SIMD unit, each core is capable of processing 16 32-bit words or 8 64-bit words per clock cycle. The cores, up to 8 memory controllers, and the PCIe client logic are connected by a logically bidirectional ring bus. Each direction of the ring consist of 5 independent rings: the data-block ring for data transfers, 2 address rings which are used to send read-/write commands and memory addresses, and 2 acknowledgment rings used to send flow control and coherence messages.

Memory accesses first go through a tag directory that checks the L2 caches of all cores for the word requested, and forwards the request to the memory controllers only if the word is not present in any L2 cache.

Current Xeon Phi cards are connected to the host system via PCIe. The Phi runs a Linux OS making it appear as a 'computer in a computer.' From the programmers point of view it can be used in 'offload mode' as a coprocessor to the host's CPU—the same way as current GPGPUs are used for computations—or else as a separate compute device in 'native mode.' In both cases, applications that execute on the Xeon Phi are scheduled by its Linux OS. In particular the Phi is capable of executing (parallel) programs written for x86 CPUs using MPI, OpenMP, or pThreads, for instance. Development cycles thus can be shortened by simply reusing x86 codes written for the CPU, (maybe) adapting them to benefit from Phi specific instructions, and recompiling them for the Xeon Phi. Throughout the entire chain, developer tools and programming languages known from x86 programming can be used.

3.2 Nvidia Fermi GPU Architecture

The Fermi GPU architecture [20, 21] is build around a scalable set of streaming multiprocessors (SM), each of which consisting of 32 scalar processors (SPs or CUDA cores), two thread schedulers, four special function units, 16 load/store units, and 64kB of on-chip low latency memory with a configurable partitioning into shared memory and L1 cache. Fermi-based GPUs are equipped with up to 16 SMs, up to 6 GB of ECC main memory (accessible with up to 177 GB/s), and are connected to the host system via PCIe.

In the CUDA programming model, GPU programs consist of a host program written in C/C++ and a set of GPU kernels which when called from within the host program are executed on the GPU asynchronously. In this model the GPU acts as a coprocessor to the host system. It is left to the programmer to partition the problem at hand into sub-problems that can be mapped onto the GPU's scalar processors by the SM's thread schedulers. The number of CUDA threads that execute the sub-problems is defined by the programmer using a grid-block hierarchy of threads.

Thread blocks are dynamically created at run time by the GPU scheduler and then are assigned to SMs for execution. The thread schedulers on the SMs partition these thread blocks into so-called warps (groups of 32 threads) which execute on the SPs in SIMD (more precisely: SIMT—Same-Instruction Multiple-Threads) manner using up to 48-way interleaved multithreading. The two thread schedulers per SM manage a total of up to 1536 concurrent threads (48 warps), distributed over up to 8 thread blocks.

During their execution, threads can access different memory layers ranging from fast on-chip shared memory (used for intra-thread-block communication/-synchronization and data sharing) to 'more or less exotic' texture memory with special data fetching modes, constant memory, and finally the global memory.

	Intel CPU Xeon E5-2680 (Sandy-Bridge)	Intel Xeon Phi Coprocessor 5110P	Nvidia GPU Tesla M2090 (Fermi)
Cores/Multi-Processors	8	60	16
Logical Core Count	16 (=8x2)	240 (=60x4)	$\leq$768 (=16x48)
SIMD Width (32-Bit)	4 (SSE), 8 (AVX)	16	$\leq$32 (SIMT)
Clock Frequency	2.7 GHz	1.05 GHz	1.3 GHz
Card Memory Size	—	8 GB (ECC)	6 GB (ECC)
Memory Bandwidth	51.2 GB/s	320 GB/s	177 GB/s
Power Consumption	$<$130 W	$<$225 W	$<$225 W

Table 1: Characteristics of the processor platforms used for simulations and benchmarking (information are from the vendors and [8, 9, 20]). The logical core count refers on CPU to the total number of hyper-threads and on Xeon Phi and GPU to the number of SIMD threads scheduled in hardware.

4. THE ISING MODEL

The Ising model postulates a periodic d-dimensional lattice with magnetic dipoles placed on the lattice sites, and with each of them associated with a spin s_i taking on values $s_i = \pm 1$. The model is described by the Hamiltonian

$$H = -\sum_{\langle ij \rangle} J_{ij} s_i s_j + h \sum_i s_i \,,$$

where $\langle ij \rangle$ refers to sites i and j are nearest neighbors ('NN'), J_{ij} is a coupling constant, and h is an external magnetic field. What makes the Ising model be of interest for various considerations is the fact that in more than one dimension it exhibits a phase transition if $h=0$. For the $(d>1, h=0)$-NN Ising model (subsequently, Ising model for short) the order parameter of the phase transition is the spontaneous magnetization m_s for temperatures $T<T_c$, where T_c is the critical temperature. For temperatures $T>T_c$ the model is in a paramagnetic phase, whereas for $T<T_c$ it is in an antiferromagnetic ($J_{ij}<0$) or in a ferromagnetic ($J_{ij}>0$) phase.

Simulating the Ising model on the computer can be done by means of Monte Carlo methods [19]. The focus is on generating a sequence of spin configurations $s^{\mu_1}, s^{\mu_2}, s^{\mu_3}, \ldots$ using some kind of update method for the spins s_i, and to take measurements over the course of the simulation. There are 2^N spin configurations s^μ with $N = \prod_{k=1}^{d} n_k$, where n_k is the lattice extent in direction k. Configuration s^μ is generated with probability $p_\mu = \frac{1}{Z}\exp(-\beta E_\mu)$, where $Z = \sum_\mu \exp(-\beta E_\mu)$ is the partition function and E_μ is the energy of the spin system occupying configuration s^μ. We here introduced the inverse temperature $\beta = 1/kT$, with k being Boltzmann's constant, and T being the system's temperature.

Common methods for generating configurations s^μ are the Metropolis algorithm, and cluster algorithms—the single-cluster algorithm by U. Wolff [27] and the multi-cluster algorithm by R. H. Swendsen and J.-S. Wang [22] are the most popular ones. The latter of the two is more suitable for parallelization as, unlike the former

one, it divides up the entire lattice into clusters. When to use the Metropolis algorithm or cluster algorithms depends on the temperature T the spin system should be simulated at. Cluster algorithms are most efficient for simulations at $T \approx T_c$. In all other cases the Metropolis algorithm is the better choice due to its simplicity. In addition to its universality for simulation purposes, the Metropolis algorithm for the Ising model also serves as an ideal candidate for benchmarking of parallel computers as its locality allows for highly parallel implementations.

The Metropolis algorithm and the multi-cluster algorithm differ in how spin configurations are altered. The discarding aspect for a configuration s^ν, created from configuration s^μ, is an increase of the system energy E, making transitions $s^\mu \rightarrow s^\nu$ with $\Delta E = E_\nu - E_\mu > 0$ be less attractive than those with $\Delta E \leq 0$. While the former of the two algorithms has its focus on changing the orientation of single spins (local update algorithm), the latter one considers clusters of spins (non-local update algorithm)—at criticality, the cluster size diverges and clusters grow up to the lattice extent.

For the Metropolis algorithm, the transition $s^\mu \rightarrow s^\nu$ is accepted with probability $p_{\mu\rightarrow\nu} = \min(1, \exp(-\beta\Delta E))$, where the computation of ΔE is as follows [19]: choose a spin s_i at random, assume this spin is flipped over, and compute $\Delta E = -2Js_i^\mu \sum_{\langle ij \rangle} s_j^\mu$. If we give up choosing spins s_i at random, but divide the lattice into even and odd sites (checkerboard decomposition), where a site is said to be even if its Cartesian coordinates add up to an even value (otherwise it is said to be odd), all even sites can be updated simultaneously in a first step, and then all odd sites are updated in a second step. Updating the entire lattice defines a 'sweep.' This approach allows for highly parallel implementations of the Metropolis algorithm.

Going from s^μ to s^ν by means of the Swendsen-Wang algorithm is done by first dividing up the entire lattice into clusters of spins, and then by flipping each of these clusters with probability one half [19]. A cluster of spins refers to a subset $\sigma_k \subseteq \{s_i \,|\, 0 \leq i \leq N-1\}$ of the spins with each spin $s_j \in \sigma_k$ being the nearest neighbor of at least one other spin $\sigma_k \ni s_j' \neq s_j$ if $|\sigma_k| > 1$, and with all spins $s_j \in \sigma_k$ having the same orientation. The subsets σ_k correspond to connected components in graph theory if the lattice is understood as a graph with vertices given by the lattice sites, and with edges connecting NN spins with the same orientation. For simulation purposes, we remove some of the edges with probability $p_{\text{delete}} = \exp(-2J\beta)$, depending on the temperature T. NN spins with the same orientation then do not necessarily belong to the same σ_k.

5. SIMULATING THE ISING MODEL

In this section we consider a squared respectively cubic lattice with periodic boundary conditions for simulation purposes.

5.1 General Aspects

When simulating the Ising model or other Ising-like spin models on parallel computers, we are usually interested in large system sizes. To meet memory limitations, information about spin orientations are stored in the smallest computer word possible. As for the Ising model $s_i = \pm 1$, we use arrays of the C data type `char` to store spins. When spins are processed, converting them to a data type other than `char` might be meaningful.

Another aspect is random number generation. As investigated in [16, 26], simple random number generators like LCG and MWC should not be used for high precision Monte Carlo simulations because of systematic errors due to deficiencies in random number sequences [7]. Unfortunately, on the GPU well established generators such as Lüscher's Ranlux [15] or the Mersenne twister random number generator [18] allow for fast random number generation just if the amount of requested random numbers is sufficiently large—on the GPU these generators consume too many resources to have each individual thread run its own instance. Making use of these generators for simulations, however, would require us to precompute random numbers, store them in main memory, and then to reload them from memory during the simulation. If lots of random numbers are consumed by the simulation, the portion of the main memory that is given to random number generation becomes non-negligible, which then in turn restricts the extent of the system to be simulated. A better approach in this respect is to produce random numbers when they are needed.

On the CPU and the Xeon Phi we found the Mersenne twister random number generator, however, be suitable for on-the-fly random number generation with acceptable production rate. For our parallel setups we use `dcmt` by M. Matsumoto and T. Nishimura [17] and a configuration file for up to 8192 independent Mersenne twisters. The suitability of `dcmt` for parallel random number generation is documented by Matsumoto and Nishimura, and for our purposes it is supported by simulation results presented in Tab. 2.

On the GPU we use a combination of an MWC and a 32-bit Xorshift generator with random state shuffling. The source code for that generator is listed in [26]. The quality of the random number sequences is demonstrated by the author, and by this paper as well.

5.2 Swendsen-Wang Multi-Cluster Algorithm

Unlike local update algorithms, such as of Metropolis type, cluster algorithms consider regions of connected aligned NN spins (clusters) to be flipped over as a whole instead of flipping single spins. As for $T \rightarrow T_c$ clusters have sizes up to the lattice extent, flipping them results in significant alterations of spin configurations. For the Ising model it is known that cluster algorithms almost entirely remove the critical slowing down problem, local update algorithms suffer from. Critical slowing down means that the relaxation time for the system increases, and finally diverges, as the system parameters are moved towards criticality.

The most popular cluster algorithms for the Ising model are the single-cluster algorithm by U. Wolff [27] and the multi-cluster algorithm by R. H. Swendsen and J.-S. Wang [22]. While the Swendsen-Wang ('SW') cluster algorithm divides up the entire lattice into clusters and then flips each of them with probability one half, the single-cluster algorithm builds up just one cluster per update step which then is flipped over. The SW algorithm therefore is the better candidate for parallelization.

The serial algorithm is depicted in Listing 1. *In words*: Iterate over all spins of the system one after another, and for each spin that does not belong to an already existing cluster start a new cluster with that spin as the root. Clusters are created as follows: Given a root, consider all of its aligned NN spins not being part of an already existing cluster for possible inclusion into the current cluster with probability $p_{\text{add}} = 1 - p_{\text{delete}}$, and repeat this procedure for all newly included spins until no more spins are included.

With respect to graph theory, clusters map to the connected components ('CC') σ_k of the graph $G = (V, E)$ with vertices $V = \{v_0, v_1, \ldots, v_{N-1}\}$ corresponding to spins $s_0, s_1, \ldots, s_{N-1}$, and edges $E = \{(v_i, v_j) \,|\, s_i \text{ and } s_j \text{ are aligned NN sites on the lattice} \wedge \text{weight } w_{ij} = 1\}$. Weights w_{ij} are 1 with probability p_{add}. Otherwise, $w_{ij} = 0$. The `findAndFlipCluster()` function (see Listing 1) determines the CCs σ_k using an adapted version of the breadth-first search (BFS) algorithm. Given a root, the exploration of the associated CC is performed in a wavefront manner. Wavefronts here are subsets of vertices that are reachable from the root, and that have the same distance to the root in terms of fewest number of edges. The wavefront exploration is enforced by the queue data structure.

```
swendsenWangUpdate(L[N])
  for i=0 to N-1 do
    if L[i] is marked as 'discovered' then
      continue
    else
      findAndFlipCluster(L,i)
  TAKE SYSTEM OBSERVABLES HERE
  for i=0 to N-1 do // reset
    mark L[i] as 'undiscovered'

findAndFlipCluster(L[N],root)
  flip=false
  draw a random number r uniform on [0,1)
  if r < 0.5 then
    flip=true
  Queue q=∅
  q.enqueue(root)
  mark L[root] as 'discovered'
  while q.empty()==false do
    x=q.dequeue()
    forall 'undiscovered' aligned NNs L[y] of L[x] do
      draw a random number r uniform on [0,1)
      if r < p_add then
        q.enqueue(y)
        mark L[y] as 'discovered'
    L[x]=L[x]*(flip==true?-1:+1) // in-place cluster flipping
```

Listing 1: Pseudo-code of the Swendsen-Wang multi-cluster algorithm for the Ising model. The system has N spins s_i stored in L[]**. The notation** predicate?a:b **is a short form of if** predicate==true **then** a **else** b**.**

On a parallel computer with possibly hundreds or thousands of cores, a parallel version of Listing 1 would suffer from parallelism can be achieved across the breadth of the wavefronts only, and also from global synchronization in the context of the wavefront execution. Other approaches to finding clusters make use of union-find data structures that are accessed by multiple concurrent threads. Investigations on porting cluster algorithms for the two-dimensional Ising model to the GPU can be found in [12, 13, 23].

5.3 Shared Memory Implementation

Our approach breaks down finding and flipping clusters into 3 steps. At first, we partition the lattice into equal-sized sub-lattices. Sub-clusters then are identified independently within sub-lattices. Since clusters might span several sub-lattices, in-place cluster flipping is not possible in this step as it is not known which sub-clusters belong to the same cluster. Instead, sub-clusters are assigned unique labels. In a second step, sub-clusters are merged together so that finally all of them belonging to the same cluster are assigned the same label—the merging problem is defined as the geometric connected component labeling (GCCL) problem [3]. In a third step, clusters are flipped over with probability one half, and data structures are prepared for a new iteration of the cluster algorithm.

Subsequently, we give some details on how these steps are implemented on the hardware used in this paper.

Step 1a: Sub-lattices are loaded into local memory, which is the shared memory on the GPU, and SIMD registers on the Xeon Phi. With respect to Sec. 5.1, spin orientations s_i are stored as 8-bit words—the values $s_i = \pm 1$ are encoded as 0 or 1 into the 1st bit of the word. When loaded, these 8-bit words are converted to 32-bit words. On the Xeon Phi we utilize the _mm512_extload_epi32() intrinsic for that purpose, whereas on the GPU we use a simple type cast.

After the load, we draw a set of random numbers in order to establish edges between aligned NN spins with probability $p_{\text{add}} = 1 - \exp(-2J\beta)$. In two (three) dimensions there are up to two

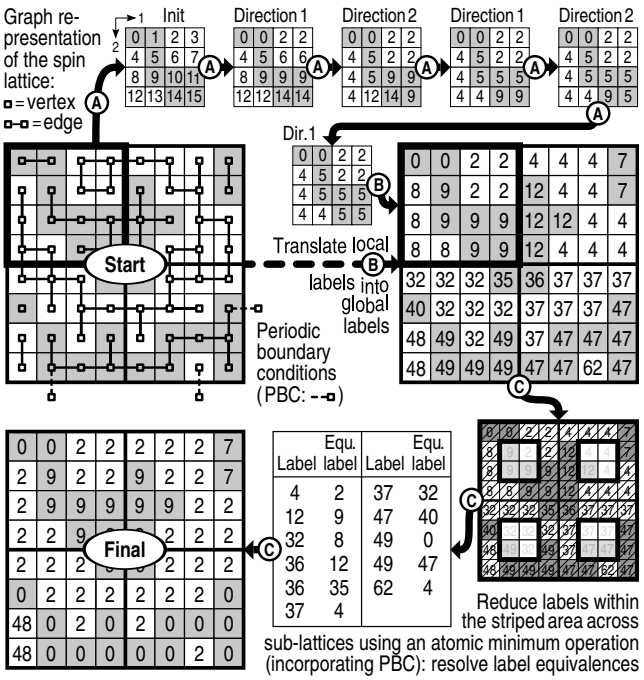


Figure 1: Illustration of the labeling for a 16×16 lattice. Step Ⓐ draws on the cluster self-labeling method by C. F. Coddington and P. D. Baillie [1]. Step Ⓑ refers to the label translation from local to global ones. After the translation, labels are reduced across the boundaries of the sub-lattices (striped area in Ⓒ) so as to finally have each cluster be represented by one label.

(three) edges e_{i_1}, e_{i_2}(and e_{i_3}) $\in E$ connecting spin s_i and its aligned nearest neighbors in positive spatial direction 1, 2 (and 3). The information on whether these edges are actually established is encoded into the 3rd, 4th (and 5th) bit of the word—these bits then are set to 1. As s_i might be the root of a cluster, we use the 2nd bit as the 'flip-bit' (see below)—it is set to 1 with probability one half.

The 8-bit words then are written to main memory with the first 5 bits set to either 0 or 1 each. On the Xeon Phi the conversion is done using the _mm512_extstore_epi32() intrinsic, whereas on the GPU we again use a type cast.

During the cluster labeling and flipping, all information encoded into the 8-bit words can be recovered using bitwise AND with 0x1, 0x2, 0x4, 0x8 (and 0x10). The lattice is stored in L[].

Step 1b: We use the cluster self-labeling method by C. F. Coddington and P. D. Baillie [1] for the identification of sub-clusters within sub-lattices. The idea is as follows (see Fig. 1, step Ⓐ—the 16×16 lattice is divided into 4 sub-lattices of extent 4×4 each. In three dimensions the method works quite similar): First, we assign unique labels to the sites $i \in \{0, \ldots, \tilde{N} - 1\}$ of the sub-lattice, where $\tilde{N}$ denotes the number of sub-lattice sites. We store the sub-lattice labels in CC[$0 \ldots \tilde{N} - 1$] (data type unsigned int) and we use the array index i as initial label. Thereafter, for each sub-lattice site, we consider its connected NN site in positive direction 1 and reduce the respective labels to the minimum of the two. If there is no connected NN, we move on to the next site. In Fig. 1, we start in the top-left corner CC[0] and iterate towards CC[$\tilde{N} - 1$]—note that we do not have periodic boundary conditions on the level of the sub-lattices. For all sites we note the number of reductions that differ from the identity operation. Afterwards, we apply the same scheme to all sub-lattice sites but now considering their connected NN sites

in positive direction 2. Again we note the number of non-identity reductions. If at the end this number is larger than zero, we reset the non-identity counter and repeat the entire reduction procedure. Otherwise, the labeling within the sub-lattice is complete, and we stop the iteration.

Obviously, this method terminates after $O(\tilde{N}=\prod_{k=1}^{d}\tilde{n}_k)$ update steps for an $\tilde{n}_d\times\tilde{n}_{d-1}\times\ldots\times\tilde{n}_1$-sub-lattice. It can be improved by some kind of 'path compression' known from grouping m distinct elements into a collection of disjoint sets using a disjoint-set data structure [4]. In our case the path compression can be achieved by replacing $\mathsf{CC}[i]=\mathsf{CC}[\mathsf{CC}[i]]$ for all $i\in\{0,\ldots,\tilde{N}-1\}$ after every sequence of reductions in positive direction $1,2,\ldots,d$—doing the replacement with multiple concurrent threads does not result in the same CC[] array as obtained by using a single thread.

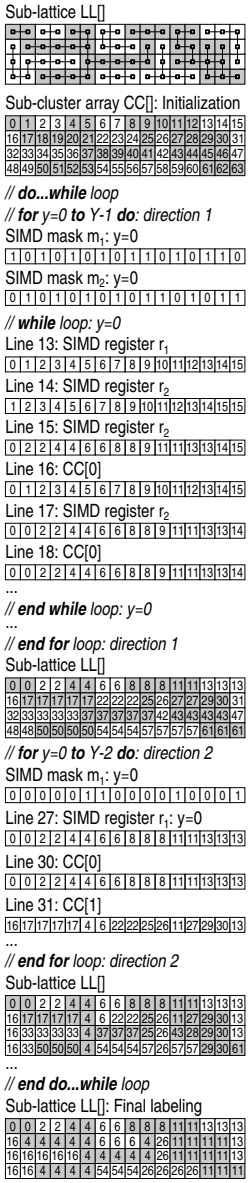
On the Xeon Phi the replacement $\mathsf{CC}[i]=\mathsf{CC}[\mathsf{CC}[i]]$ can be implemented as $\mathsf{CC}[i]=$ _mm512_i32gather_epi32(CC[i],&CC[0],4)$_{i=0,16,32,\ldots}$ if the array CC[] is 64-byte aligned and has size $(\prod_{k=2}^{d}\tilde{n}_k)\times 16$. $\tilde{n}_1=16$ follows from array entries in direction 1 are direct successors in main memory, and the Phi's SIMD width for 32-bit words is 16.

```
selfLabeling(LL[],CC[])
  // Requirements:
  //  LL[] is ñ2x16 sub-lattice, ñ2∈ℕ
  //  CC[] is ñ2x16 sub-cluster array, ñ2∈ℕ
  //  LL[],CC[] 64-byte aligned
  do
    changes=0
    // direction 1
    for y=0 to ñ2-1 do
      i=16*y
      m1=vecCmp(vecAnd(LL[i],0x4),0x4,EQ);
      m2=m1≪1
      while true do
13:     r1=CC[i]
14:     r2=vecPermute((1,2,..,13,14,15,15)16,r1)
15:     r2=vecMaskMin(r2,m1,r1,r2)
16:     vecMaskStore(&CC[i],m1,r2)
17:     r2=vecPermute((0,0,1,2,3,..,13,14)16,r2)
18:     CC[i]=vecMaskMin(CC[i],m2,CC[i],r2)
        if vecCmp(CC[i],r1,NE)==0 then
          break
        else
          changes++
    // direction 2
    for y=0 to ñ2-2 do
      i=16*y
      m1=vecCmp(vecAnd(LL[i],0x8),0x8,EQ)
27:   r1=vecMin(CC[i],CC[i+1])
      changes+=vecMaskCmp(m1,CC[i],r1,NE)
      changes+=vecMaskCmp(m1,CC[i+1],r1,NE)
30:   vecMaskStore(&CC[i],m1,r1)
31:   vecMaskStore(&CC[i+1],m1,r1)
    while changes>0
```

Listing 2: Pseudo-code of the cluster self-labeling method within sub-lattices using Xeon Phi SIMD intrinsics.

vecCmp: _mm512_cmp_epi32
vecMaskCmp: _mm512_mask_cmp_epi32
vecPermute: _mm512_permutevar_epi32
vecMin: _mm512_min_epi32
vecMaskMin: _mm512_mask_min_epi32
vecMaskStore: _mm512_mask_store_epi32
EQ: _MM_CMPINT_EQ, NE: _MM_CMPINT_NE

Figure 2: Illustration of the cluster self-labeling method.



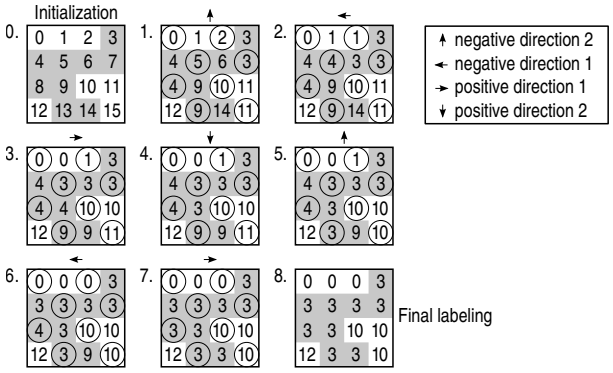


Figure 3: Cluster self-labeling method on GPU using a checkerboard decomposition of the sub-lattice. For the sake of simplicity, it is assumed that for aligned NN sites (the equal colored ones) edges are established with probability 1. Circled sites are those threads are assigned to.

Listing 2 illustrates some details of the implementation of the cluster self-labeling method in two dimensions using Xeon Phi specific SIMD intrinsics. $(x_0,x_1,\ldots,x_{15})_{16}$ denotes a 16-element SIMD register with entries x_i. SIMD masks are extracted from the bit representation of the words stored in LL[] (see Step 1a). The return value of the _mm512_[XXX]_cmp_epi32() intrinsic can be interpreted as 16-bit unsigned integer with bits set to 1 for elements for which the comparison operation evaluates to true, and 0 otherwise. For reasons of clarity and comprehensibility we introduced abbreviations for the intrinsics and point out the first steps of the iterations graphically (see Fig. 2).

As can be seen from the final label assignment, for the root $\hat{c}_k$ of cluster k it holds that $\mathsf{CC}[\hat{c}_k]=\hat{c}_k$ (for instance: $\mathsf{CC}[0]=0$, $\mathsf{CC}[2]=2$, etc.). Hereafter, we want to refer to this property as 'root-property.' After the labeling within the sub-lattices, local labels have to be translated into global ones so that all labels are still unique, and local root-labels become global root-labels. Global labels are stored in C[] of data type unsigned int.

Remark 1: On the GPU we actually make use of the path compression method, whereas on the Phi we found it slow down the simulation.

Remark 2: On the GPU there is no support for per-thread SIMD operations as the programming model addresses SIMD operations on the level of the warps. We therefore assign sub-lattices to thread-blocks of 256 threads each. This allows for a total of 1024 threads per SM if the register usage per thread is restricted to 32 (there are no limitations by the shared memory). Sub-lattice sizes on the GPU are 16×32 in two dimensions, and $4\times4\times32$ in three dimensions, respectively. The number of sites per sub-lattice is twice the number of threads per block as our implementation uses a checkerboard decomposition of the sub-lattices with one thread per even site. In addition to reducing labels in positive direction 1, 2 (and 3), label reductions then have to be performed for NNs in negative direction 1, 2 (and 3) as well. The self-labeling method on GPU is illustrated in Fig. 3 for two dimensions.

Remark 3: On Xeon Phi and CPU we use a single thread per sub-lattice. Sub-lattice sizes on the Phi are 8×16 in two dimensions, and $2\times4\times16$ in three dimensions—threads iterate over the set of sub-lattices. On the CPU there is just one sub-lattice per thread. Since our CPU's SIMD registers allow for 4 32-bit integers only (AVX has insufficient integer support), and also because the instruction set does not allow for masked SIMD operations, we have no SIMD

```
atomicMin(volatile *address,desired)
  while true do
    current=*address
    if current ≤ desired then
      return *address
    if atomicCAS(address,current,desired)==true then
      return current
```

Listing 3: Pseudo-code of an atomic minimum function. The atomic compare-and-swap function atomicCAS() **returns** true **if the value pointed to by** address **equals the value of** current**. If so, the value pointed to by** address **is atomically set to the value of** desired**. Otherwise, the** atomicCAS() **function returns** false **and the value pointed to by** address **remains the same.**

version of the labeling on CPU. Instead we leave vectorization to the compiler (the Intel compiler actually reports about the relevant loops have been vectorized).

Step 2: The merging of the sub-clusters is done by reducing the labels C[] of all aligned NN sites along the boundaries of adjacent sub-lattices (see the striped area in Fig. 1, Step ©), so that finally every cluster is represented by the smallest label of each of the sub-clusters it is made up of.

If the label reduction is done by multiple concurrent threads, some of them will find label equivalences which later might be modified by other threads. Suppose that two threads A and B find label equivalences (c_1, c_2^A) and (c_1, c_2^B) with $c_2^A < c_1$, $c_2^B < c_1$, and $c_2^A \neq c_2^B$. Since A does not know about (c_1, c_2^B), and B does not know about (c_1, c_2^A), we need a mechanism that prevents loosing information if both A and B update the label c_1 concurrently, that is, establish the reference $\mathsf{C}[c_1] = c_2^{A|B}$.

We can distinguish two cases (we assume c_1 has not changed yet, and no other thread tries to update c_1—this will be dropped below):

1) If $c_2^A < c_2^B$, we actually have label equivalences (c_1, c_2^A) and (c_2^B, c_2^A), that is, we have to establish the references
$$\mathsf{C}[c_1] = c_2^A,\ \mathsf{C}[c_2^B] = c_2^A.$$
2) If $c_2^A > c_2^B$, consider case 1) and interchange c_2^A and c_2^B.

The scheme can be easily generalized to more than two threads.

The challenging part of the update is the atomicity requirement such that a thread establishing a reference is not interrupted by any other thread in doing so. For that purpose we use an atomic minimum function which performs the update if and only if the label c_1 to be updated is larger than the desired one, say, c_2, and if it has not changed when it comes to actually establishing the reference $\mathsf{C}[c_1] = c_2$. The atomic minimum function 'fails' if in the meantime another thread has found the equivalence (c_1, c_2') with $c_2' \leq c_2$. The calling thread obtains $c_3 = c_2'$ as return value, and then has to establish the reference $\mathsf{C}[c_2] = c_3$ if $c_2 \neq c_3$. Otherwise, nothing has to be done. The update was 'successful' if the atomic minimum function returns a label $c_3 > c_2$. The calling thread then has to establish the reference $\mathsf{C}[c_3] = c_2$.

The label reduction method using the atomic minimum function is illustrated in Listing 4.

Atomic minimum function: For our GPU implementation we utilize the built-in atomicMin() available for CUDA capable Nvidia GPUs with compute capability 1.1 and above [21]. On Xeon Phi and CPU there is no such operation, so we had to build our own. Listing 3 gives the pseudo-code of our implementation.

Step 3: As a result of Step 2, we have a list of label equivalences (references) with some entries pointing to themselves—these labels are the root-labels. As our initial labels have been the array indices

```
reduceLabels(C[N])
  parallel forall NNs i and j belonging to adjacent sub-lattices do
  || c1=C[i] // label within sub-lattice A
  || while c1 ≠ C[c1] do
  ||   c1=C[c1]
  || c2=C[j] // label within sub-lattice B
  || while c2 ≠ C[c2] do
  ||   c2=C[c2]
  || while true do
  ||   c3=atomicMin(&C[c1],c2)
  ||   if c3==c2 then
  ||     break
  ||   elseif c3 > c2 then c1=c3
  ||   elseif c3 < c2 then c1=c2, c2=c3
```

Listing 4: Pseudo-code of the label reduction method using an atomic minimum function. It is assumed that sub-lattices have been already labeled, with the respective labels stored in C[]**.**

$i \in \{0, 1, \ldots, N-1\}$, the root-property implies: $\mathsf{C}[i] = \hat{c}_k \Rightarrow \hat{c}_k = i$ for all root labels $\hat{c}_k$. The information about whether to flip a cluster or not is stored in $\mathsf{L}[\hat{c}_k]$ if $\hat{c}_k$ is the root-label that represents the cluster. If for each site i we follow the references down to the root, we get the array index $\hat{c}_k = \mathsf{root}(i)$ of the root and then look up the desired information in L[]. Remember: for each lattice site i the bitwise AND of L[root(i)] and 0x2 allows to test the flip-bit. If it is set, spin s_i is flipped over using a bitwise XOR of L[i] and 0x1.

Finding out the root-label for every spin s_i can be done as follows (implementation of the root() function):

Method A: We split up the array C[] containing N cluster labels into P equal-sized contiguous partitions of size N/P each (for the sake of simplicity, we assume that N is a multiple of P). With P threads it takes $\lceil \log_2 P \rceil$ iterations to transform the array C[] into C'[] with $\mathsf{C'}[i] = \hat{c}_k$ and $\hat{c}_k$ being the root-label of the cluster s_i belongs to. During each iteration, all threads iterate through their partition and replace $\mathsf{C}[i] = \mathsf{C}[\mathsf{C}[i]]$ for $i \in [\,\mathrm{id}\frac{N}{P}, (\mathrm{id}+1)\frac{N}{P}) \cap \mathbb{N}$, where 'id' is the thread ID.

Method B: The root-label for every spin s_i is determined from scratch by traversing the array C[] from start position $c_i = \mathsf{C}[i]$, and by replacing $c_i = \mathsf{C}[c_i]$ as long as $c_i \neq \mathsf{C}[c_i]$. The method can be improved by replacing $\mathsf{C}[i] = \hat{c}_k$ after having found the root-label $\hat{c}_k$ associated with the cluster s_i belongs to ('path compression'). This way, traversals of C[] can be shortened thereafter.

The two methods are illustrated in Listing 5, including flipping the clusters.

5.4 Distributed Labeling using MPI

If lattice sizes become significantly larger than 32768×32768 or 1024×1024×1024 (values are for 6 GB Tesla M2090 and 8 GB Xeon Phi, respectively, with cluster labels stored as 32-bit unsigned integers, and spins/flip-bits/edges encoded into 8-bit words), or if the effective update time per spin saturates, the use of more than one compute device becomes necessary.

Subsequently, we describe how to extend the shared memory implementation of the SW algorithm detailed in Sec. 5.3 to make use of multiple devices. The description is for the Xeon Phi and the CPU. The approach should apply to multi-GPU systems too.

Step 0a: Divide up the lattice into equal sized slices (we assume a homogeneous compute system) and assign these slices to different MPI ranks. In our implementation, we cut the lattice along direction 2 in two dimensions, and along direction 3 in three dimensions. Within the slices the multithreaded shared memory SW algorithm is used for the labeling. The implementation in Sec. 5.3 has not to

```
flipClusters_A(L[N],C[N])
  parallel region: P threads
  || id=get thread ID unique in {0,1,...,P-1}
  || start=id*⌈N/P⌉
  || stop=min(N,start+⌈N/P⌉)
  || iterations=⌈log2 P⌉
  || // find root-labels
  || for n=1 to iterations do
  ||   for i=start to stop-1 do
  ||     C[i]=C[C[i]]
  ||   barrier
  || // flip clusters
  || for i=start to stop-1 do
  ||   if (L[C[i]]&0x2)==0x2 then
  ||     L[i]^=0x1 // flip spin : bitwise XOR

flipClusters_B(L[N],C[N])
  parallel region: P threads
  || id=get thread ID unique in {0,1,...,P-1}
  || start=id*⌈N/P⌉
  || stop=min(N,start+⌈N/P⌉)
  || for i=start to stop-1 do
  ||   c=C[i]
  ||   while c ≠ C[c] do
  ||     c=C[c]
  ||   C[i]=c // root label found: 'path compression'
  ||   if (L[c]&0x2)==0x2 then
  ||     L[i]^=0x1 // flip spin : bitwise XOR
```

Listing 5: Pseudo-code of the cluster flipping method using two different approaches.

be modified except for there are no periodic boundary conditions in direction d for the d-dimensional model.

Labels used within the slices can range from 0 to $\hat{N}-1$ if there are $\hat{N}$ sites per slice.

Step 0b: Before the actual label reduction, we translate per-slice labels into global ones (similar to the shared memory implementation). Simultaneously, we minimize the number of labels assigned to all clusters in all slices. For the minimization we determine the number of root-labels numLabels for every slice. This can be done by multiple threads concurrently. Each thread is assigned a contiguous partition of the slice in which it counts the number of root-labels. A single thread then sums up these values into numLabels and calls MPI_Scan(&numLabels,&labelOffset,...). The accumulated number of root-labels per slice labelOffset then becomes available to all ranks by:

```
PER MPI RANK DO:
  numLabelsPartition[P]
  parallel region: P Threads
  || id=get thread ID unique in {0,1,...,P-1}
  || numLabelsPartition[id]=0
  || forall sites i in partition do
  ||   if C[i] is root-label then
  ||     numLabelsPartition[id]++
  numLabels=0
  for i=0 to P-1 do
    numLabels+=numLabelsPartition[i]
  MPI_Scan(&numLabels,&labelOffset,1,UNSIGNED,MPI_SUM)
  labelOffset-=numLabels
```

The value of the labelOffset variable is used to translate per-slice labels into global labels, i.e.:

```
PER MPI RANK DO:
  sliceLabels[]
  parallel region: P Threads
  || id=get thread ID unique in {0,1,...,P-1}
  || label=0
  || for i=0 to id-1 do
  ||   label+=numLabelsPartition[i]
  || forall sites i in partition do
  ||   if C[i] is root-label then          // encoding: bit 0..30 for label
  ||     if (L[i]&0x2)==0x2 then           //           bit 31 for flip-info
11:||       C[i]=(labelOffset+label)|0x80000000  // set flip-bit : bitwise OR
  ||     else
13:||       C[i]=(labelOffset+label)|0x0        // do not set flip-bit : bitwise OR
14:||     sliceLabels[label++]=C[i]
  ||     L[i]|=0x80 // store root information in highest bit of L[i] : bitwise OR
  || barrier
  || // update cluster labels of non-root sites
  || forall sites i in partition do
  ||   if C[i] is not root-label: (L[i]&0x80)!=0x80 then
  ||     C[i]=C[C[i]]
```

In Line 11 (and 13, respectively) per-slice root-labels are replaced by (labelOffset+label)|X, where label is a per-thread label counter and X encodes whether the associated cluster should be flipped over (set bit 31) or not (do not set bit 31). Afterwards, all non-root sites in all slices adapt their new labels.

By the integration of the flip-bit into the cluster labels, all information about which clusters should be flipped and which not is automatically distributed across all ranks when labels are reduced (see below). The actual labels can be recovered by applying a bitwise AND operation with 0x7FFFFFFF. The effect of the procedure on the labeling is illustrated in Fig. 4 in two dimensions.

As we use the upper most bit as the flip-bit, the maximum number of labels that can be assigned to clusters is $2^{31}-1$ if 32-bit words are used. Although we loose a factor 2 for the labeling, for the critical Ising model the mean cluster size is significantly larger than 1. As a consequence even lattices with more than $2^{32}-1$ (largest 32-bit integer value) sites can be simulated—with the shared memory version of the SW algorithm the largest lattice is restricted to have no more than $2^{32}-1$ sites as otherwise we would not be able to represent the largest possible label with 32-bit words.

All labels assigned to the clusters are stored in the arrays sliceLabels[] (line 14) of data type unsigned int. Each rank has its own array and is responsible for a specific non-empty contiguous subset of the labels. To access a label C[i] in the sliceLabels[] array, the value of the respective labelOffset variable (each rank has its own) has to be subtracted from C[i] (see Fig. 5).

Step 1a: Labels are reduced in negative direction d. Since boundary regions connecting neighboring slices are located in the main memory of different MPI ranks, we first need to send the data from rank i to rank $(i+1) \bmod \hat{P}$ if there are $\hat{P}$ ranks (see Fig. 6).

To minimize the amount of data to be transferred, we actually send only elements for which there is an edge in positive direction d. For each such element we send its array index with respect to the

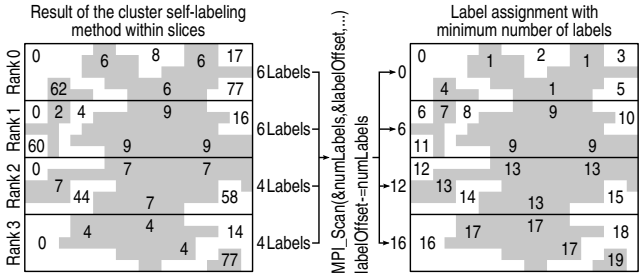


Figure 4: Preliminary step of the multi-device SW algorithm. The total number of labels used during the self-labeling within the slices is reduced so that afterwards the number of labels assigned to clusters is minimal.

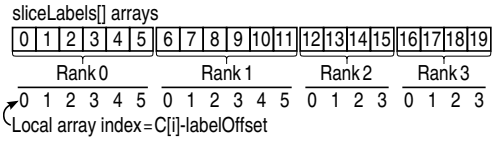

sliceLabels[] arrays
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Rank 0
Rank 1
Rank 2
Rank 3
0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 0 1 2 3
Local array index=C[i]-labelOffset

Figure 5: Relation between global labels C[i] **and the local array indices within the** sliceLabels[] **arrays.**

boundary sub-array, and the element itself, that is, 8 bytes. Since edges are established with probability $p_{\text{add}} = 1 - \exp(-2J\beta)$ if and only if the involved spins both are aligned (2 cases out of 4), the expected number of elements to be transferred is $0.29\bar{N}$ in two dimensions and $0.18\bar{N}$ in three dimensions, where $\bar{N}$ is the number of border sites in this step. That is, the expected amount of data to be send is $8{\times}0.29\bar{N}$ bytes, and $8{\times}0.18\bar{N}$ bytes, respectively. Both values are below $4{\times}\bar{N}$ bytes, which is the size of the entire border.

After the border exchange, each rank resolves label equivalences similar to the shared memory case. The only difference is that ranks can establish references only for those labels they are the owner of. These references have effect on the sliceLabels[] arrays. Reductions that affect labels the rank, say, i, is not responsible for are noted and will be send to rank $(i-1+\hat{P}) \bmod \hat{P}$ in Step 2.

Similar to the shared memory case, we also note the number of non-identity reductions.

Step 1b: Same as Step 1a, but labels are reduced in positive direction d. Again the number of non-identity reductions is noted.

After the reductions, we restart with Step 1a if any labels have been actually reduced. Otherwise, we continue with Step 2.

Step 2: For each site i in each slice determine the root-label and apply a bitwise AND with 0x80000000. If the upper most bit is set, flip the respective spin using bitwise XOR of L[i] and 0x1.

In our actual implementation we further minimize the number of transferred labels by sending information about label changes only. The procedure is illustrated in Listing 6 using pseudo-code.

6. CODE VALIDATION AND BENCHMARKING

In this section we summarize 1) the observables computed with the two- and the three-dimensional Ising model simulations at criticality using our implementations, and 2) the measured runtimes on the many-core processors.

6.1 Validation—Computed Observables

For the two-dimensional Ising model the inverse critical temperature is $T_c^{-1} = \frac{1}{2}\log(1+\sqrt{2})$ [10], whereas in three dimensions we use $T_c^{-1} \approx 0.22165$ [2] and $T_c^{-1} \approx 0.22165455$ [5], respectively.

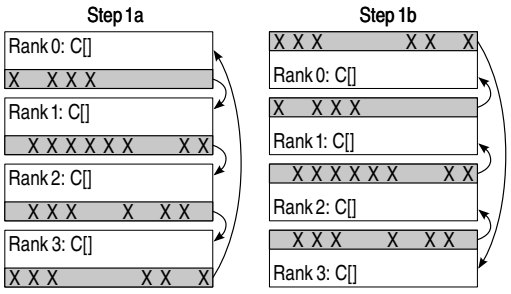


Figure 6: Border exchange of the C[] **array. The** Xs **mark array entries that are actually transferred.**

```
flipClusters_MPI(L[],C[])
 PER MPI RANK DO: P̂ Ranks
  rank=get rank ID unique in {0,1,...,P̂-1}
  parallel for i=0 to X-1 do
   if (L[N̂-X+i]&0x8)==0x8 then // edge in direction 2
    label=root(C[N̂-X+i])
    cBorderU[i]=label
    changesU[] ← (i,label)
   else
    cBorderU[i]=0xFFFFFFFF // invalid label: no edge present
   cBorderL[i]=0xFFFFFFFF // invalid label: default value
  parallel region: P Threads
  || id=get thread ID unique in {0,1,...,P-1}
  || start=id*⌈X/P⌉
  || stop=min(X,start+⌈X/P⌉)
  || do
  ||  barrier
  ||  if id==0 then // Step 1a
  ||   size=2*getSize(changesU)
  ||   MPI_Isend(changesU,size,UNSIGNED,(rank+1)%P̂,1)
  ||   MPI_Probe((rank-1+P̂)%P̂,1,status)
  ||   size=status.getCount()
  ||   MPI_Recv(changesL,size,UNSIGNED,(rank-1+P̂)%P̂,1,status)
  ||   APPLY CHANGES TO cBorderL[]
  ||   changesSlice=0
  ||  changesL[]=∅
  ||  barrier
  ||  for i=start to stop-1 do
  ||   if cBorderL[i]!=0xFFFFFFFF then
  ||    label=root(C[i]) // the root function uses sliceLabels[]
30:||   if (cBorderL[i]&0x7FFFFFFF)>(label&0x7FFFFFFF) then
L1:||    cBorderL[i]=label
  ||    changesL[] ← (i,label)
  ||    atomicAdd(&changesSlice,1)
  ||   elseif (cBorderL[i]&0x7FFFFFFF)<(label&0x7FFFFFFF) then
  ||    if labelOffset≤(label&0x7FFFFFFF) then // my labels
  ||     ADAPT sliceLabels[] USING ATOMIC MINIMUM OPERATION
37:||    IF NECESSARY, GOTO L1
  ||  barrier
  ||  if id==0 then // Step 1b
  ||   size=2*getSize(changesL)
  ||   MPI_Isend(changesL,size,UNSIGNED,(rank-1+P̂)%P̂,2)
  ||   MPI_Probe((rank+1)%P̂,2,status)
  ||   size=status.getCount()
  ||   MPI_Recv(changesU,size,UNSIGNED,(rank+1)%P̂,2,status)
  ||   APPLY CHANGES TO cBorderU[]
  ||  changesU[]=∅
  ||  barrier
  ||  for i=start to stop-1 do
  ||   if cBorderU[i]!=0xFFFFFFFF then
  ||    label=root(C[N̂-X+i]) // the root function uses sliceLabels[]
  ||    LINES 30..37 WITH 'xxxU' AND 'xxxL' INTERCHANGED
  ||  barrier
  ||  if id==0 then
  ||   MPI_Allreduce(&changesSlice,&changes,1,UNSIGNED,MPI_SUM)
  ||  barrier
  || while changes>0
```

Listing 6: Pseudo-code of the label reduction according to Step 1a,b using $\hat{P}$ ranks. We consider the two-dimensional case with slices having $\hat{N}$ sites and extent $Y/\hat{P} \times X$. The lattice itself has extent $Y \times X$. barrier means a thread barrier.

We evaluate the observables 'internal energy' E and 'specific heat' $c_V = \frac{\partial E}{\partial T}|_{V=\text{const}}$, each per spin, using the Monte Carlo estimates

$$E_{\text{MC}} = \frac{1}{N}\frac{1}{M}\sum_{m=1}^{M} E_{\mu_m} ,$$

$$(c_V)_{\text{MC}} = \frac{1}{N}\frac{1}{T_c^2}\left(\frac{1}{M}\sum_{m=1}^{M} E_{\mu_m}^2 - \left(\frac{1}{M}\sum_{m=1}^{M} E_{\mu_m}\right)^2\right).$$

N is the system volume in terms of spins (see Sec. 5.3), M is the number of measurements taken during the simulation, and E_{μ_m} is the internal energy of the system in configuration $s^{\mu m}$.

From Tab. 2 it can be seen that in the case of the two-dimensional Ising model Monte Carlo estimates are in agreement with exact calculations according to A. E. Ferdinand and M. E. Fisher [6]—Monte Carlo estimates of E_{MC} and $(c_V)_{\mathrm{MC}}$ are correct within errors up to the 6th, and the 2nd/3rd decimal place, respectively. In three dimensions, we compared estimates for $L \in \{32, 64, 128\}$ at $T_c^{-1} = 0.22165$ with values measured by Barber et al. [2]—for the comparison, our values of the internal energy and the specific heat need to be multiplied by $1/3$ respectively $T_c^2/3$. We found agreement within errors. To our knowledge, there is no published work giving estimates of E and c_V for $L \gg 128$ in three dimensions.

Error estimation was done by means of the Γ-method [28], incorporating autocorrelation times.

For selected setups, we performed multiple runs of the simulation to check the reproducibility of the measurements. We found exact agreement for all runs. Since simulation results additionally match exact values and values from literature, we assume our implementations be correct.

Remark: The specific heat c_V diverges as $T \to T_c$. Since the estimate 0.22165455 is closer to the actual value of T_c^{-1} than is 0.22165, c_V values are larger for $T_c^{-1} \approx 0.22165455$. Additionally, increasing the lattice extent brings us more close to the infinite volume limit, also resulting in c_V increases.

6.2 Hardware Configurations and Parallel Programming APIs

Xeon Phi and CPU benchmarks have been performed on a cluster with 16 compute nodes provided to us by Intel. Each node hosts 2 Intel Xeon E5-2680 CPUs (octa-core Sandy-Bridge, 2.7 GHz) and 2 Intel Xeon Phi coprocessor 5110P. Nodes are equipped with 128 GB DDR3 RAM and run Red Hat Linux 6.3 with IBM Platform HPC 3.2 and kernel version 2.6.32. They are connected using 2 single-port Intel True Scale QDR Infiniband cards. The Intel MPSS stack version is 2.1.6720-13 update 3. All Xeon Phi cards are configured to directly communicate with each other via Infiniband.

The GPU benchmarks ran on a Supermicro server with X8DTG-QF+ motherboard, two Intel Xeon E5620 CPUs (quad-core Westmere, 2.4 GHz), 48 GB DDR3 RAM, and four Nvidia Tesla M2090 GPU modules in PCIe x16 slots each. The system runs a Scientific Linux 6.1 with kernel version 2.6.38.

On Xeon Phi and CPU we use OpenMP for parallelization. Vectorization on the Phi is done using intrinsics. On the CPU we leave vectorization to the compiler for two reasons: first, SSE/AVX intrinsics do not support masking operations, which we use intensively on the Phi, and second, AVX has no full integer support.

Program code is compiled with Intel's `icpc-13.1.3` to Xeon Phi 'native' executables and executables for the CPU, respectively. On the GPU Nvidia's CUDA 4.0 API and `nvcc` compiler is used.

6.3 Benchmark Results

For benchmarking purposes on CPU and Xeon Phi, we use as many threads as there are logical compute cores on the hardware. Single- and dual-socket setups on CPU thus use 16 respectively 32 threads with Hyper-Threading enabled. Threads are pinned to cores using `KMP_AFFINITY=compact,granularity=fine`. On the Xeon Phi we use 240 threads pinned to cores via `KMP_AFFINITY= balanced,granularity=fine`. On both the Xeon Phi and the CPU we found these setups give best performance. For GPU benchmarks the number of threads is adapted to the lattice extents, which throughout all benchmarks are chosen such that the core count on the device is matched—on the Xeon Phi lattice extents thus are not necessarily a power of 2.

Xeon Phi	d=2, $T_c^{-1}=\frac{1}{2}\log(1+2^{1/2})$			
L	$-E_{\mathrm{Exact}}$	$-E_{\mathrm{MC}}$	$(c_v)_{\mathrm{Exact}}$	$(c_v)_{\mathrm{MC}}$
480	1.4155103...	1.415516(8)	3.1909689...	3.186(3)
960	1.4148619...	1.414862(5)	3.5339349...	3.531(5)
1920	1.4145377...	1.414535(4)	3.8768120...	3.869(8)
3840	1.4143756...	1.414373(3)	4.2196445...	4.234(11)
7680	1.4142946...	1.4142963(17)	4.5624548...	4.575(15)
15360 (MPI)	1.4142540...	1.4142544(14)	4.9052539...	4.89(2)
30720 (MPI)	1.4142338...	1.414233(3)	5.2480475...	5.31(8)

Xeon Phi	d=3, T_c^{-1}=0.22165		d=3, T_c^{-1}=0.22165455	
L	$-E_{\mathrm{MC}}$	$(c_v)_{\mathrm{MC}}$	$-E_{\mathrm{MC}}$	$(c_v)_{\mathrm{MC}}$
32	1.00696(4)	2.234(4)	—	—
64	0.996582(17)	2.705(4)	—	—
128	0.992642(8)	3.202(5)	—	—
256	0.991134(8)	3.645(16)	0.991479(8)	3.867(17)
512	0.990516(8)	3.78(5)	0.990932(10)	4.54(6)
1024	0.990332(6)	3.47(9)	0.990726(8)	5.37(14)

Tesla M2090	d=2, $T_c^{-1}=\frac{1}{2}\log(1+2^{1/2})$			
L	$-E_{\mathrm{Exact}}$	$-E_{\mathrm{MC}}$	$(c_v)_{\mathrm{Exact}}$	$(c_v)_{\mathrm{MC}}$
512	1.4154292...	1.415430(8)	3.2229079...	3.222(4)
1024	1.4148214...	1.414812(5)	3.5658628...	3.563(5)
2048	1.4145174...	1.414518(5)	3.9087343...	3.906(12)
4096	1.4143655...	1.414363(4)	4.2515641...	4.248(18)
8192	1.4142895...	1.414290(3)	4.5943730...	4.61(3)

Tesla M2090	d=3, T_c^{-1}=0.22165		d=3, T_c^{-1}=0.22165455	
L	$-E_{\mathrm{MC}}$	$(c_v)_{\mathrm{MC}}$	$-E_{\mathrm{MC}}$	$(c_v)_{\mathrm{MC}}$
32	1.00698(4)	2.234(3)	—	—
64	0.99659(2)	2.698(5)	—	—
128	0.992650(9)	3.208(7)	—	—
256	0.991133(7)	3.652(14)	0.991461(10)	3.83(3)
512	0.990543(7)	3.82(5)	0.990975(11)	4.63(6)
1024	0.990328(4)	3.46(5)	0.990718(6)	5.17(10)

Table 2: Monte Carlo estimates E_{MC} and $(c_V)_{\mathrm{MC}}$, each per spin. In two dimensions we use $T_c^{-1} = \frac{1}{2}\log(1+\sqrt{2})$. In three dimensions we use $T_c^{-1} = 0.22165$ and $T_c^{-1} = 0.22165455$. Exact calculations in two dimensions are according to A. E. Ferdinand and M. E. Fisher.

For MPI benchmarks using the CPU we create one MPI rank per socket and use 16 threads per rank. For MPI benchmarks using the Phi, we create one rank per device and use 240 threads per rank. Network communication goes over Infiniband. We use Intel's MPI implementation (version 4.1.1) with fabrics `shm:dapl` on CPU, and `shm:tmi` on Xeon Phi.

Runtime measurements are done using `omp_get_wtime()` (OpenMP timer). The accuracy of the timer is 'nano-seconds.' Since we measure the execution time of thousands of lattice updates and then compute update times per spin, we found almost no variations in runtime measurements for multiple runs using the same setup.

Execution times per spin update are summarized in Table 3, and illustrated in Fig. 7. The parallel CPU runtimes are obtained with 16 (single-socket) and 32 (dual-socket) threads, respectively. It can be seen that for sufficiently large lattices the GPU and the Xeon Phi lie almost at level, with the Xeon Phi a little faster.

Both accelerators achieve speedups over the multithreaded single-socket CPU version of about a factor 3. If both sockets are used, the speedup breaks down to a little more than a factor 1.5. Execution

Xeon Phi	d=2, $T_c^{-1}=\frac{1}{2}\log(1+2^{1/2})$				d=3, $T_c^{-1}=0.22165455$			
L	$t_{CPU}^{Sequential}$	$t_{CPU}^{Single\text{-}Sckt.}$	$t_{CPU}^{Dual\text{-}Sckt.}$	$t_{Xeon\ Phi}$	$t_{CPU}^{Sequential}$	$t_{CPU}^{Single\text{-}Sckt.}$	$t_{CPU}^{Dual\text{-}Sckt.}$	$t_{Xeon\ Phi}$
32	—	—	—	—	32.2	3.86	3.05	5.70
64	27.6	4.69	6.15	19.0	31.4	3.34	2.02	2.30
128	27.3	3.30	2.66	7.03	32.1	3.55	1.84	1.66
240	29.6	3.08	2.05	3.62	33.4	4.21	2.17	1.30
480	29.3	2.93	1.64	1.66	33.5	4.71	2.58	1.33
960	29.3	2.89	1.52	1.12	33.7	4.80	2.61	1.41
1920	29.5	3.30	1.63	1.04	—	—	—	—
3840	30.1	3.42	1.81	1.01	—	—	—	—
7680	30.1	3.46	1.84	1.02	—	—	—	—

Tesla M2090	d=2, $T_c^{-1}=\frac{1}{2}\log(1+2^{1/2})$				d=3, $T_c^{-1}=0.22165455$			
L	$t_{CPU}^{Sequential}$	$t_{CPU}^{Single\text{-}Sckt.}$	$t_{CPU}^{Dual\text{-}Sckt.}$	t_{M2090}	$t_{CPU}^{Sequential}$	$t_{CPU}^{Single\text{-}Sckt.}$	$t_{CPU}^{Dual\text{-}Sckt.}$	t_{M2090}
32	—	—	—	—	32.2	3.86	3.05	3.75
64	27.6	4.69	6.15	10.7	31.4	3.34	2.02	1.92
128	27.3	3.30	2.66	3.72	32.1	3.55	1.84	1.52
256	27.3	2.95	1.92	2.09	32.2	4.18	2.16	1.47
512	27.3	2.80	1.58	1.52	32.4	4.62	2.58	1.49
1024	27.3	2.78	1.45	1.31	33.1	4.78	2.59	1.54
2048	27.4	3.24	1.60	1.21	—	—	—	—
4096	27.5	3.26	1.75	1.18	—	—	—	—
8192	27.5	3.29	1.79	1.16	—	—	—	—

Comparison against known GPU implementations: d=2, $T_c^{-1}=\frac{1}{2}\log(1+2^{1/2})$

L	Weigel [23] t_{GPU}	Komura [13] t_{GPU}	Wende et al. (this paper) t_{M2090}	$t_{Xeon\ Phi}$
256	—	5.47 (GTX580)	2.09	3.62 (L=240)
512	6.533 (GTX480)	3.54 (GTX580)	1.52	1.66 (L=480)
1024	—	2.98 (GTX580)	1.31	1.12 (L=960)
2048	—	2.86 (GTX580)	1.21	1.04 (L=1920)
4096	—	2.87 (GTX580)	1.18	1.01 (L=3840)
8192	3.934 (Tesla M2070)	—	1.16	1.02 (L=7680)
8192	2.697 (GTX580)	—	1.16	1.02 (L=7680)

Table 3: Update time per spin in nano-seconds for the simulation of the two- and the three-dimensional critical Ising model on an L^d-lattice. Single-socket and dual-socket CPU setups use 16 respectively 32 threads. On Xeon Phi 240 threads are used. On GPU the thread count adapts to the lattice extent.

times per spin for the three-dimensional model are slightly larger than their counterparts in two dimensions, which is due to a larger number of arithmetic operations per spin, and also due to less favorable memory access patterns compared to the two-dimensional case.

A direct comparison of our GPU implementation with implementations by Y. Komura and Y. Okabe [13], and M. Weigel [23] is also given in Tab. 3. Our codes achieve more than a factor 2 performance gain over known GPU implementations. We want to emphasize that almost all runtimes in [13, 23] are obtained with consumer GPUs. In the case of the GTX580 the theoretical peak performance is about a factor 1.15 higher than for the Tesla M2090. The Tesla M2090 thus is at a disadvantage in this comparison, and effective performance gains of our implementations are larger than stated.

Figure 8 illustrates the strong scaling of our implementations with the number of threads P on Xeon Phi, and the number of thread blocks P^* (of size 256 each) on GPU. Execution times are found to scale linearly with P and P^* as long as the number of threads is below or equal to the number of logical execution units on the hardware. However, the scaling is not ideal. One reason for this is that the label reduction does not ideally scale with the number of threads. Since the propagation of label equivalences from

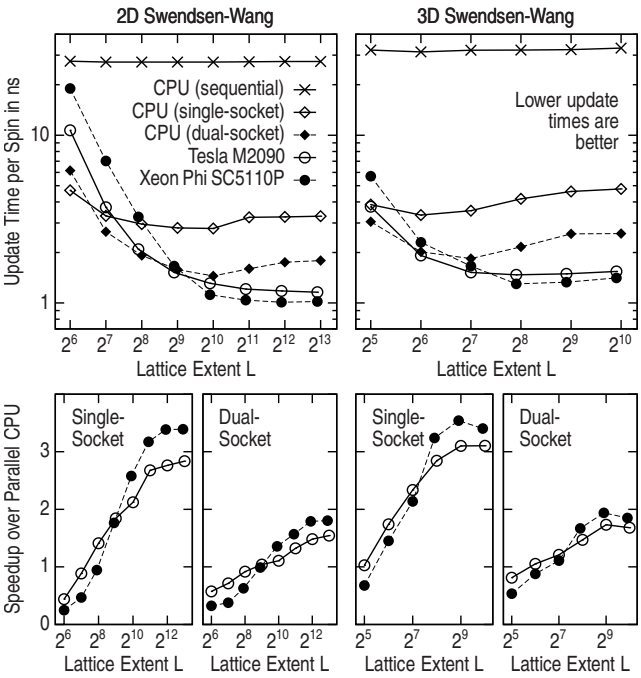


Figure 7: Update times per spin (*top*) and speedups over a parallelized CPU version of the SW algorithm (*bottom*).

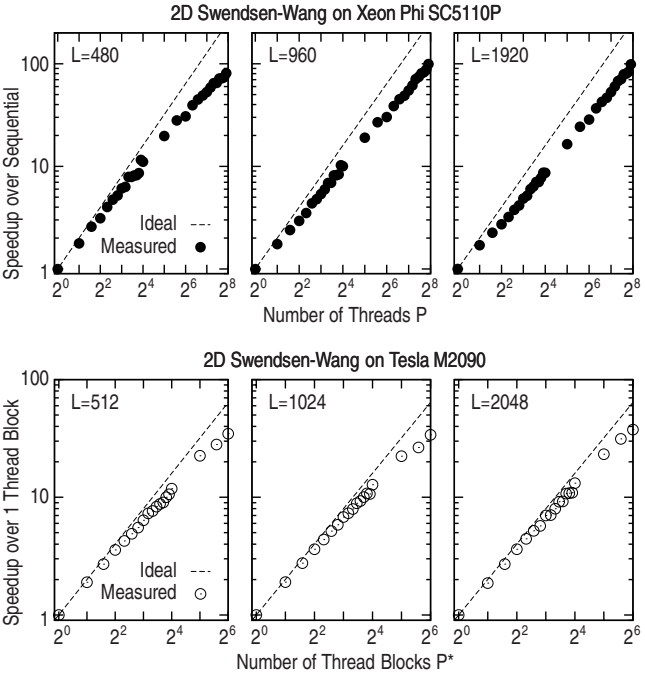


Figure 8: Strong scaling of the implementations of the SW algorithm with the number of threads P on Xeon Phi (*top*) and the number of thread blocks P^* on the GPU (*bottom*).

the larger ones to the lower ones is done by multiple concurrent threads, some of them interfere with each other. A certain amount of work thus is done several times. With increasing number of threads, the effect on the execution time becomes significant.

Table 4 summarizes update times per spin using MPI. Due to the MPI overhead in our implementations (including algorithmic modifications), an execution with one MPI rank is significantly slower

Xeon Phi	d=2, $T_c^{-1}=\frac{1}{2}\log(1+2^{1/2})$					
L	$t_{1\text{-Rank}}$	$t_{2\text{-Ranks}}$	$t_{4\text{-Ranks}}$	$t_{8\text{-Ranks}}$	$t_{16\text{-Ranks}}$	$t_{32\text{-Ranks}}$
15360	1.791	0.902	0.460	0.249	0.181	0.201
30720	1.779	0.896	0.454	0.236	0.145	0.124
61440	—	—	0.489	0.259	0.145	0.075
128880	—	—	—	—	0.132	0.070

CPU	d=2, $T_c^{-1}=\frac{1}{2}\log(1+2^{1/2})$					
L	$t_{1\text{-Rank}}$	$t_{2\text{-Ranks}}$	$t_{4\text{-Ranks}}$	$t_{8\text{-Ranks}}$	$t_{16\text{-Ranks}}$	$t_{32\text{-Ranks}}$
16384	4.858	2.434	1.237	0.632	0.343	0.216
32768	4.856	2.433	1.228	0.625	0.324	0.182
65536	5.199	2.624	1.316	0.664	0.338	0.180
131072	—	—	1.435	0.720	0.362	0.182

Xeon Phi	d=3, T_c^{-1}=0.22165455					
L	$t_{1\text{-Rank}}$	$t_{2\text{-Ranks}}$	$t_{4\text{-Ranks}}$	$t_{8\text{-Ranks}}$	$t_{16\text{-Ranks}}$	$t_{32\text{-Ranks}}$
960	2.285	1.151	0.649	0.393	0.234	0.159
1440	—	—	0.632	0.335	0.204	0.129
1920	—	—	—	0.317	0.178	0.107
2400	—	—	—	—	0.169	0.092

CPU	d=3, T_c^{-1}=0.22165455					
L	$t_{1\text{-Rank}}$	$t_{2\text{-Ranks}}$	$t_{4\text{-Ranks}}$	$t_{8\text{-Ranks}}$	$t_{16\text{-Ranks}}$	$t_{32\text{-Ranks}}$
1024	6.900	3.462	1.734	0.907	0.494	0.332
1536	6.886	3.448	1.723	0.881	0.472	0.275
2048	—	3.460	1.741	0.884	0.457	0.253
2560	—	—	1.785	0.877	0.454	0.248

Table 4: Executions times per spin update in nano-seconds for the simulation of the two- and the three-dimensional critical Ising model on an L^d-lattice using MPI with up to 16 compute nodes and 2 CPU sockets respectively 2 Xeon Phis per node.

than its shared memory counterpart. We thus deem it meaningful to consider just cases in which update times per spin using the shared memory version saturate, and/or the lattice becomes too large to hold it in the main memory of a single Xeon Phi card.

Update times per spin are illustrated in Fig. 9. For large lattices, the available amount of main memory on the Xeon Phi and the CPU restricts the lower number of MPI ranks used for the benchmark.

It can be seen that the update times reduce almost linearly with the number of MPI ranks if lattices are sufficiently large. With 32 Xeon Phis we achieve about $\frac{1}{0.07\text{ns}} \approx 14.3$ and $\frac{1}{0.092\text{ns}} \approx 10.9$ spin flips per nano-second in two respectively three dimensions on lattices with extent 128880×128880 and $2400\times2400\times2400$.

Our CPU implementation using MPI performs about a factor 2.6 below its Xeon Phi counterpart in two dimensions, and about a factor 2.7 below it in three dimensions. The scaling with the number of ranks is almost linear, even on smaller lattices.

Efficiency considerations with the shared memory version as a reference attest our Xeon Phi MPI version a poor performance, as we obtain only a speedup of about $\frac{1.02\text{ns}}{0.070\text{ns}} \approx 14.6$ in two and $\frac{1.41\text{ns}}{0.092\text{ns}} \approx 15.3$ in three dimensions using 32 Xeon Phis. The efficiency thus is $\approx 45\%$. Taking one MPI process as a reference—which appears more meaningful to us, as the MPI version is algorithmically more complex than the shared memory version—, the efficiency is above 75% in two and three dimensions. On the CPU the efficiency is above 80% (with respect to the MPI execution using one process). Currently, all network traffic between two Xeon Phi cards installed in two different nodes first goes over the PCIe bus and then over the actual network. With the next Intel MIC architecture standalone devices become available and the coprocessor-host bottleneck should disappear. We assume that with an improved Phi-to-Phi network

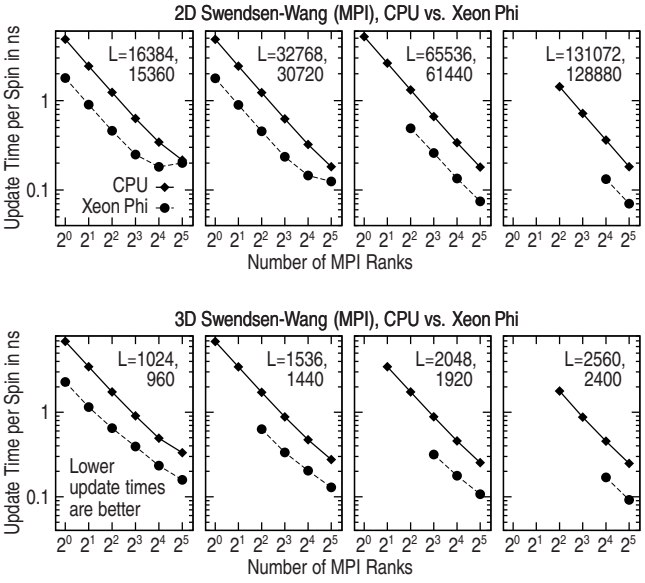


Figure 9: Update times per spin in nano-seconds (MPI version of the SW algorithm). For some setups the available amount of main memory restricts the minimum number of ranks.

connection our implementations' efficiencies become larger than 75% on the Xeon Phi.

7. SUMMARY AND CONCLUSION

In this paper we present parallel implementations of Swendsen and Wang's multi-cluster algorithm for the Ising model using current parallel processor platforms—Intel Xeon Phi coprocessor 5110P, Nvidia Tesla M2090 GPU, and Intel octa-core Xeon E5-2680.

Our approach for shared memory machines draws on a decomposition of the lattice containing the Ising spins into sub-lattices, and parallel cluster labeling within these sub-lattices by means of C. F. Coddington and P. D. Baillie's [1] cluster self-labeling method. The label reduction across sub-clusters is novel in that atomic hardware primitives are used to resolve label equivalences. The reduction process is improved by providing information about label equivalences to all concurrent threads during the merging by means of 'path compression.'

Our codes have been validated by comparing Monte Carlo estimates of selected observables against exact calculations [6] in two dimensions, and against literature values in three dimensions [2]. Agreement within statistical errors could be noticed.

We found both the Xeon Phi and the GPU give measurable performance gains over comparable parallel implementations executed on the CPU. Speedups up to a factor 3 can be observed. For simulation setups with sufficiently large lattices, the Xeon Phi lies level with the GPU.

A direct comparison with known implementations of the Swendsen-Wang algorithm for the two-dimensional Ising model on GPU [13, 23] and multi-GPU [14] setups demonstrates our codes are more than a factor 2 faster. In three dimensions, no references using current computer hardware could be found.

Our MPI version of the Swendsen-Wang algorithm abstracts the shared memory version by dividing the lattice into slices which are assigned to MPI ranks and then are labeled using our shared memory approach. Afterwards we perform an intermediate step that minimizes the total number of labels used across all slices—this way, the simulation of lattices with more than $2^{32} - 1$ sites be-

comes possible despite 32-bit words are used for the labeling. Label equivalences on the level of the slices are resolved by exchanging borders between ranks that work on neighboring slices as long as labels change. Hereby, each rank is the owner of a subset of the labels, and for these labels it establishes label references which finally allow for each site to find the cluster it belongs to. The border exchange is improved by actually sending information about label changes instead of entire borders.

The integration of MPI functionalities into our codes was exactly the same for both Xeon Phi and CPU. We thus had to implement it just once. On both the Phi and the CPU we achieve almost linear scaling with the number of MPI ranks if lattices are sufficiently large. The 32-rank Phi execution using 240 threads per rank gives performance gains more than a factor 2.5 over a 32-rank CPU execution using 16 threads per rank (Hyper-Threading enabled).

A valuable feature of our implementations using MPI is that lattices with more than $2^{32}-1$ sites can be simulated with 32-bit words used for the labeling—we found our implementations be suitable for lattices even larger than 131072×131072 respectively $2560\times2560\times2560$.

We experienced programming the Xeon Phi to be straightforward. Many programming techniques, especially vectorization by means of intrinsics, could be adopted from x86 CPU programming. The development of the codes took us a few days, starting with the CPU codes and then by applying changes to make them run on the Xeon Phi. For the Xeon Phi the integration of SIMD intrinsics consumed the major portion of the time.

As our GPU codes are quite similar to the Xeon Phi codes, except for replacing SIMD operations by SIMT operations (on the level of the warps), the development here also took us a few days only.

Acknowledgment

This work is partially funded by the German BMBF project ENHANCE, grant no. 01IH11004G. We thank Intel for generously providing us Xeon Phi hardware accelerators for code development, and for access to a Xeon Phi cluster for benchmarking. In particular, in-depth discussions with Dr. Michael Klemm shed light on details of the Intel MIC architecture.

8. REFERENCES

[1] C. F. Baillie and P. D. Coddington. Cluster identification algorithms for spin models - sequential and parallel, 1991.

[2] M. N. Barber, R. B. Pearson, D. Toussaint, and J. L. Richardson. Finite-size scaling in the three-dimensional Ising model. *Phys. Rev. B*, 32:1720–1730, Aug 1985.

[3] K. P. Belkhale and P. Banerjee. Parallel algorithms for geometric connected component labeling on a hypercube multiprocessor. *IEEE Transactions on Computers*, 41:699–709, 1992.

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[5] Y. Deng and H. W. J. Blöte. Simultaneous analysis of several models in the three-dimensional Ising universality class. *Phys Rev E Stat Nonlin Soft Matter Phys*, 68(3 Pt 2):036125, 2003.

[6] A. E. Ferdinand and M. E. Fisher. Bounded and inhomogeneous Ising models. I. Specific-heat anomaly of a finite lattice. *Phys. Rev.*, 185(2):832–846, Sept. 1969.

[7] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from 'good' random number generators. *Physical Review Letters*, 69(23):3382+, 1992.

[8] A. Heinecke, M. Klemm, and H. J. Bungartz. From GPGPU to many-core: Nvidia Fermi and Intel Many Integrated Core architecture. *Computing in Science and Engineering*, 14:78–83, 2012.

[9] Intel. Intel Xeon Phi coprocessor 5110P, product brief, 2012.

[10] E. Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik A Hadrons and Nuclei*, 31(1):253–258, Feb. 1925.

[11] J. Jeffers, J. R. Jeffers, and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier Science & Technology Books, 2013.

[12] Y. Komura and Y. Okabe. GPU-based single-cluster algorithm for the simulation of the Ising model. *J. Comput. Phys.*, 231(4):1209–1215, Feb. 2012.

[13] Y. Komura and Y. Okabe. GPU-based Swendsen-Wang multi-cluster algorithm for the simulation of two-dimensional classical spin systems. *Computer Physics Communications*, 183(6):1155–1161, 2012.

[14] Y. Komura and Y. Okabe. Multi-GPU-based Swendsen-Wang multi-cluster algorithm for the simulation of two-dimensional q-state potts model. *Computer Physics Communications*, 184(1):40 – 44, 2013.

[15] M. Lüscher. A portable high quality random number generator for lattice field theory simulations. *Comput. Phys. Commun.*, 79:100–110, 1994.

[16] M. Manssen, M. Weigel, and A. K. Hartmann. Random number generators for massively parallel simulations on GPU. (arXiv:1204.6193), Apr 2012.

[17] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generators. pages 56–69, June 1998.

[18] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, Jan. 1998.

[19] E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Clarendon Press, 1999.

[20] Nvidia. Fermi compute architecture whitepaper, v1.1, 2009.

[21] Nvidia. *Nvidia CUDA C programming guide, v4.0*. 2011.

[22] R. H. Swendsen and J.-S. Wang. Nonuniversal critical dynamics in Monte Carlo simulations. *Phys. Rev. Lett.*, 58:86–88, Jan 1987.

[23] M. Weigel. Connected component identification and cluster update on GPU. (arXiv:1105.5804), May 2011.

[24] M. Weigel. Simulating spin models on GPU. *Computer Physics Communications*, 182(9):1833–1836, 2011.

[25] M. Weigel. Performance potential for simulating spin models on GPU. *J. Comput. Physics*, 231(8):3064–3082, 2012.

[26] F. Wende. Master thesis: Simulation of spin models on Nvidia graphics cards using CUDA, 2010.

[27] U. Wolff. Comparison between cluster Monte Carlo algorithms in the Ising model. 1989.

[28] U. Wolff. Monte Carlo errors with less errors. *Comput. Phys. Commun.*, 156:143–153, 2004.